The roadmap

The aim is to understand this:

> *In category theory, the concept of catamorphism denotes the unique homomorphism from an initial algebra into some other algebra.*

To better understand this:

> *In functional programming, catamorphisms provide generalizations of folds of lists to arbitrary algebraic data types, which can be described as initial algebras. The dual concept is that of anamorphism that generalize unfolds. A hylomorphism is the composition of an anamorphism followed by a catamorphism.*

The outline

- Basic category theory

The outline

- Basic category theory
- Haskell as a category (if you squint)

The outline

- Basic category theory
- Haskell as a category (if you squint)
- Category theoretic functors, Haskell Functors endofunctors

The outline

- Basic category theory
- Haskell as a category (if you squint)
- Category theoretic functors, Haskell Functors endofunctors
- Algebras over an endofunctor

The outline

- Basic category theory
- Haskell as a category (if you squint)
- Category theoretic functors, Haskell Functors endofunctors
- Algebras over an endofunctor
- F-Algebra homomorphisms, or arrows in the category of F-algebras

The outline

- Basic category theory
- Haskell as a category (if you squint)
- Category theoretic functors, Haskell Functors endofunctors
- Algebras over an endofunctor
- F-Algebra homomorphisms, or arrows in the category of F-algebras
- Initial objects

The outline

- Basic category theory
- Haskell as a category (if you squint)
- Category theoretic functors, Haskell Functors endofunctors
- Algebras over an endofunctor
- F-Algebra homomorphisms, or arrows in the category of F-algebras
- Initial objects
- Catamorphisms as unique homomorphisms from initial objects

The outline

- Basic category theory
- Haskell as a category (if you squint)
- Category theoretic functors, Haskell Functors endofunctors
- Algebras over an endofunctor
- F-Algebra homomorphisms, or arrows in the category of F-algebras
- Initial objects
- Catamorphisms as unique homomorphisms from initial objects
- Flip the arrows

What's a category?

Pick some things:

- Objects (X,Y,Z)

Assert some properties:

Commutative diagram:

$$hom \circ alg \equiv alg' \circ fmap\ hom$$

What's a category?

Pick some things:

- Objects (X,Y,Z)
- Arrows between objects (f,g,$g \circ f$)

Assert some properties:

Commutative diagram:

$$hom \circ alg \equiv alg' \circ fmap\ hom$$

What's a category?

Pick some things:

- Objects (X,Y,Z)
- Arrows between objects (f,g,$g \circ f$)

Assert some properties:

- All arrows compose associatively

Commutative diagram:

$$hom \circ alg \equiv alg' \circ fmap\ hom$$

What's a category?

Pick some things:

- Objects (X,Y,Z)
- Arrows between objects (f,g,$g \circ f$)

Assert some properties:

- All arrows compose associatively
- Every object has an identity arrow

Commutative diagram:

$$hom \circ alg \equiv alg' \circ fmap\ hom$$

What's a category?

Pick some things:

- Objects (X,Y,Z)
- Arrows between objects (f,g,$g \circ f$)

Assert some properties:

- All arrows compose associatively
- Every object has an identity arrow

Commutative diagram:

$$hom \circ alg \equiv alg' \circ fmap\ hom$$

What's a category?

Pick some things:

- Objects (X,Y,Z)
- Arrows between objects (f,g,$g \circ f$)

Assert some properties:

- All arrows compose associatively
- Every object has an identity arrow

Commutative diagram:

$$hom \circ alg \equiv alg' \circ fmap\ hom$$

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[loop left, "id_X"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \arrow[loop right, "id_Y"] \\
& Z \arrow[loop below, "id_Z"]
\end{tikzcd}$$

Figure 1:

Haskell as sort of a category

Pick some things:

- Objects are types (not values!)

Assert some properties:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
id :: a -> a
```

NB: it's lies, all lies!

```
seq undefined () = undefined
seq (undefined . id) () = ()
```

Haskell as sort of a category

Pick some things:

- Objects are types (not values!)
- Arrows are functions between types

Assert some properties:

```haskell
(.) :: (b -> c) -> (a -> b) -> a -> c
id :: a -> a
```

NB: it's lies, all lies!

```haskell
seq undefined () = undefined
seq (undefined . id) () = ()
```

Haskell as sort of a category

Pick some things:

- Objects are types (not values!)
- Arrows are functions between types

Assert some properties:

- All arrows compose associatively

```
(.) :: (b -> c) -> (a -> b) -> a -> c
id :: a -> a
```

NB: it's lies, all lies!

```
seq undefined () = undefined
seq (undefined . id) () = ()
```

Haskell as sort of a category

Pick some things:

- Objects are types (not values!)
- Arrows are functions between types

Assert some properties:

- All arrows compose associatively
- Every object has an identity arrow

```
(.) :: (b -> c) -> (a -> b) -> a -> c
id :: a -> a
```

NB: it's lies, all lies!

```
seq undefined () = undefined
seq (undefined . id) () = ()
```

Functors, endofunctors

A functor is a mapping between categories that sends objects to objects (types to types) and arrows to arrows (terms to terms), preserving identity arrows and composition, possibly across categories.

```
fmap id = id
fmap f . fmap g = fmap (f . g)
```

Endofunctors map from a category to the same category.

In the case of Hask:

```
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

Algebra "over" an endofunctor

For a category C and endofunctor F an algebra of F is an object X in C and a morphism:

$$alg : F(X) \to X$$

X is called the "carrier" of the algebra.

```haskell
-- For a category and endofunctor
data F a = Zero | Succ a

instance Functor F where
  fmap _ Zero     = Zero
  fmap f (Succ a) = Succ (f a)

-- An algebra of F is an X in C
type X = Natural

-- And a morphism
alg :: F X -> X
alg Zero = 0
alg (Succ n) = n + 1
```

F Natural -> Natural

```haskell
data F a = Zero | Succ a

alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1


> alg Zero
0
> alg $ Succ $ alg Zero
1
> alg $ Succ $ alg $ Succ $ alg Zero
2
```

An alternate algebra, same F and C, different X

```
alg' :: F String -> String
alg' Zero     = "!"
alg' (Succ s) = "QUACK" ++ s

> alg' $ Succ $ alg' $ Succ $ alg' Zero
"QUACKQUACK!"
```

Prescient fun fact

> *An initial object of a category C is an object I in C such that for every object X in C, there exists precisely one morphism $I \to X$. - Wikipedia*

(up to isomorphism)

Initial, in the category of algebras

An initial algebra for an endofunctor F on a category C is an initial object in the category of algebras of F.

Initial, in the category of algebras

An initial algebra for an endofunctor F on a category C is an initial object in the category of algebras of F.

Category of algebras of F:

- Objects: alg, alg', ...

Initial, in the category of algebras

An initial algebra for an endofunctor F on a category C is an initial object in the category of algebras of F.

Category of algebras of F:

- Objects: alg, alg', ...
- Arrows : structure preserving maps (homomorphisms) from an algebra to another

Homomorphisms between two algebras.

An arrow in the category of F-algebras of a given endofunctor e.g. between (Natural, alg) and (String, alg') is a function mapping the carrier in the underlying category (Hask, hom : Natural -> String), such that the following square commutes:

$$\begin{tikzcd}
F(Natural) \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\
Natural \arrow[r, "hom"'] & String
\end{tikzcd}$$

Figure 2:

That is to say that

```haskell
fNat :: F Natural
fNat = Succ 1

hom :: Natural -> String
hom n = timesN n "QUACK" ++ "!"

> alg fNat              -- 2
> fmap hom fNat         -- Succ "QUACK!"
> hom $ alg fNat        -- "QUACKQUACK!"
> alg' $ fmap hom fNat -- "QUACKQUACK!"
```

$$hom \circ alg \equiv alg' \circ fmap\ hom$$

$$\begin{tikzcd}
F(Natural) \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\
Natural \arrow[r, "hom"'] & String
\end{tikzcd}$$

Figure 3:

Initial, in the category of algebras

An initial algebra for an endofunctor F on a category C is an initial object in the category of algebras of F.

Category of algebras of F:

- Objects: alg, alg', ...

Initial, in the category of algebras

An initial algebra for an endofunctor F on a category C is an initial object in the category of algebras of F.

Category of algebras of F:

- Objects: alg, alg', ...
- Arrows : structure preserving maps (homomorphisms) from an algebra to another

Initial, in the category of algebras

An initial algebra for an endofunctor F on a category C is an initial object in the category of algebras of F.

Category of algebras of F:

- Objects: alg, alg', . . .
- Arrows : structure preserving maps (homomorphisms) from an algebra to another

Initial, in the category of algebras

An initial algebra for an endofunctor F on a category C is an initial object in the category of algebras of F.

Category of algebras of F:

- Objects: alg, alg', ...
- Arrows : structure preserving maps (homomorphisms) from an algebra to another

Initial:

- There is a unique morphism from the initial algebra to *all other algebras*.

What fits?

- The carrier must not "lose" any information, or there is some algebra that it cannot map to.

$$X \textit{ initial} \implies F(X) \cong X$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\
InitF \arrow[r, "hom"'] & String
\end{tikzcd}$$

Figure 4:

What fits?

- The carrier must not “lose” any information, or there is some algebra that it cannot map to.
- The carrier can’t add information, or the morphism won’t be unique.

$$X \text{ initial} \implies F(X) \cong X$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\
InitF \arrow[r, "hom"'] & String
\end{tikzcd}$$

Figure 4:

What fits?

- The carrier must not "lose" any information, or there is some algebra that it cannot map to.
- The carrier can't add information, or the morphism won't be unique.
- The algebra must have type: F InitF -> InitF

$$X \ \mathit{initial} \implies F(X) \cong X$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\
InitF \arrow[r, "hom"'] & String
\end{tikzcd}$$

Figure 4:

What fits?

- The carrier must not "lose" any information, or there is some algebra that it cannot map to.
- The carrier can't add information, or the morphism won't be unique.
- The algebra must have type: F InitF -> InitF
- Lambek's theorem says that if there is an initial object, it is isomorphic to the carrier via the algebra

$$X \ \mathit{initial} \implies F(X) \cong X$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\
InitF \arrow[r, "hom"'] & String
\end{tikzcd}$$

Figure 4:

What fits?

- The carrier must not "lose" any information, or there is some algebra that it cannot map to.
- The carrier can't add information, or the morphism won't be unique.
- The algebra must have type: F InitF -> InitF
- Lambek's theorem says that if there is an initial object, it is isomorphic to the carrier via the algebra
- data InitF = InitF (F InitF)

$$X \ \textit{initial} \implies F(X) \cong X$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\
InitF \arrow[r, "hom"'] & String
\end{tikzcd}$$

Figure 4:

More generally. . .

More generally...

```haskell
data Fix f = Roll { unRoll :: f (Fix f) }
type InitF = Fix F
Roll :: F InitF -> InitF
unRoll :: InitF -> F InitF
```

More generally. . .

```
data Fix f = Roll { unRoll :: f (Fix f) }
type InitF = Fix F
Roll :: F InitF -> InitF
unRoll :: InitF -> F InitF
```

```
fix2 :: InitF
fix2 = Roll $ Succ $ Roll $ Succ $ Roll Zero
```

More generally. . .

```
data Fix f = Roll { unRoll :: f (Fix f) }
type InitF = Fix F
Roll :: F InitF -> InitF
unRoll :: InitF -> F InitF
```

```
fix2 :: InitF
fix2 = Roll $ Succ $ Roll $ Succ $ Roll Zero
```

If this is the initial object in the category of algebras, there must be a unique arrow from InitF to *every* algebra:

$$\forall algebras\ \exists hom\ : InitF \to carrier\ of\ algebra$$

The unique homomorphism

$$\forall algebras\ \exists hom\ : InitF \to carrier\ of\ algebra$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] \arrow[d, "Roll"'] & F(Natural) \arrow[d, "alg"] \\
InitF \arrow[r, "hom"'] & Natural
\end{tikzcd}$$

Figure 5:

The unique homomorphism

$$\forall algebras\ \exists hom\ : InitF \rightarrow carrier\ of\ algebra$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] \arrow[d, "Roll"'] & F(Natural) \arrow[d, "alg"] \\
InitF \arrow[r, "hom"'] & Natural
\end{tikzcd}$$

Figure 5:

```
Roll :: F InitF -> InitF
```

The unique homomorphism

$$\forall \mathit{algebras}\ \exists \mathit{hom}\ : \mathit{InitF} \rightarrow \mathit{carrier\ of\ algebra}$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] \arrow[d, "Roll"'] & F(Natural) \arrow[d, "alg"] \\
InitF \arrow[r, "hom"'] & Natural
\end{tikzcd}$$

Figure 5:

```
Roll :: F InitF -> InitF
```

```
hom :: InitF -> Natural
```

The unique homomorphism

$$\forall \textit{algebras}\ \exists \textit{hom} \ : \textit{InitF} \to \textit{carrier of algebra}$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] & F(Natural) \arrow[d, "alg"] \\
InitF \arrow[u, "unRoll"] \arrow[r, "hom"'] & Natural
\end{tikzcd}$$

Figure 6:

```
unRoll :: InitF -> F InitF
```

The unique homomorphism

$$\forall \textit{algebras}\ \exists \textit{hom} : \textit{InitF} \to \textit{carrier of algebra}$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] & F(Natural) \arrow[d, "alg"] \\
InitF \arrow[u, "unRoll"] \arrow[r, "hom"'] & Natural
\end{tikzcd}$$

Figure 6:

```haskell
unRoll :: InitF -> F InitF
```

```haskell
hom :: InitF -> Natural
hom = alg . fmap hom . unRoll
```

The unique homomorphism

$$\forall algebras\ \exists hom\ : InitF \rightarrow carrier\ of\ algebra$$

$$\begin{tikzcd}
F\ InitF \arrow[r, "F(hom)"] & F(Natural) \arrow[d, "alg"] \\
InitF \arrow[u, "unRoll"] \arrow[r, "hom"'] & Natural
\end{tikzcd}$$

Figure 6:

```haskell
unRoll :: InitF -> F InitF
```

```haskell
hom :: InitF -> Natural
hom = alg . fmap hom . unRoll
```

```haskell
cata :: Functor f
     => (f a -> a) -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unRoll

hom = cata alg
```

Evaluation of cata

```haskell
cata :: Functor f
     => (f a -> a) -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unRoll
```

```haskell
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

Evaluation of cata

```haskell
cata :: Functor f
     => (f a -> a) -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unRoll
```

```haskell
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

```haskell
cata alg (Roll $ Succ $ Roll Zero)
```

Evaluation of cata

```haskell
cata :: Functor f
     => (f a -> a) -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unRoll
```

```haskell
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

```haskell
cata alg (Roll $ Succ $ Roll Zero)
```

```haskell
alg $ fmap (cata alg) (Succ $ Roll Zero)
```

Evaluation of cata

```haskell
cata :: Functor f
     => (f a -> a) -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unRoll
```

```haskell
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

```haskell
cata alg (Roll $ Succ $ Roll Zero)
```

```haskell
alg $ fmap (cata alg) (Succ $ Roll Zero)
```

```haskell
alg $ Succ $ cata alg $ Roll Zero
```

Evaluation of cata

```haskell
cata :: Functor f
     => (f a -> a) -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unRoll
```

```haskell
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

```haskell
cata alg (Roll $ Succ $ Roll Zero)
```

```haskell
alg $ fmap (cata alg) (Succ $ Roll Zero)
```

```haskell
alg $ Succ $ cata alg $ Roll Zero
```

```haskell
alg $ Succ $ alg $ fmap (cata alg) Zero
```

Evaluation of cata

```haskell
cata :: Functor f
     => (f a -> a) -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unRoll
```

```haskell
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

```haskell
cata alg (Roll $ Succ $ Roll Zero)
```

```haskell
alg $ fmap (cata alg) (Succ $ Roll Zero)
```

```haskell
alg $ Succ $ cata alg $ Roll Zero
```

```haskell
alg $ Succ $ alg $ fmap (cata alg) Zero
```

```haskell
alg $ Succ $ alg $ Zero
```

This is recursion in a general sense

```
data Nat a = Succ a | Zero
```

This is recursion in a general sense

```
data Nat a = Succ a | Zero
```

```
data String a = Cons Char a | End
```

This is recursion in a general sense

```
data Nat a = Succ a | Zero
```

```
data String a = Cons Char a | End
```

```
data BinaryTree a = Branch a a | Tip
```

This is recursion in a general sense

```haskell
data Nat a = Succ a | Zero
```

```haskell
data String a = Cons Char a | End
```

```haskell
data BinaryTree a = Branch a a | Tip
```

```haskell
data RoseTree a = Branches [a] | Tip
```

This is recursion in a general sense

```
data Nat a = Succ a | Zero
```

```
data String a = Cons Char a | End
```

```
data BinaryTree a = Branch a a | Tip
```

```
data RoseTree a = Branches [a] | Tip
```

```
data Group a = Action a a | Inv a | Unit
```

Hutton's razor - final tagless

```
class Calculator a where
    lit :: Int -> a
    add :: a -> a -> a
    mult :: a -> a -> a

instance Calculator Int where
    lit = id
    add = (+)
    mult = (*)

instance Calculator String where
    lit = show
    add s1 s2 = s1 ++ " + " ++ s2
    mult s1 s2 = s1 ++ " x " ++ s2
```

Hutton's razor - F algebra

```
data Calculator a = Lit Int | Add a a | Mult a a deriving Functor

evalAlg :: Calculator Int -> Int
evalAlg (Lit i) = i
evalAlg (Add i1 i2) = i1 + i2
evalAlg (Mult i1 i2) = i1 * i2

ppAlg :: Calculator String -> String
ppAlg (Lit i) = show i
ppAlg (Add s1 s2) = s1 ++ " + " ++ s2
ppAlg (Mult s1 s2) = s1 ++ " x " ++ s2

pp :: Fix Calculator -> String
pp = cata ppAlg
```

Damn the torpedos, flip the arrows

```
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

Damn the torpedos, flip the arrows

```haskell
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

```haskell
coalg :: Natural -> F Natural
coalg 0 = Zero
coalg n = Succ (n - 1)
```

Damn the torpedos, flip the arrows

```haskell
alg :: F Natural -> Natural
alg Zero = 0
alg (Succ n) = n + 1
```

```haskell
coalg :: Natural -> F Natural
coalg 0 = Zero
coalg n = Succ (n - 1)
```

For a category C and endofunctor F a co-algebra of F is an object X in C and a morphism:

$$coalg : X \rightarrow F(X)$$

Morphisms on coalgebras: co all of the things!

$$
\begin{tikzcd}
F(Natural) \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\
Natural \arrow[r, "hom"'] & String
\end{tikzcd}
$$

Figure 7:

Morphisms on coalgebras: co all of the things!

$$\begin{tikzcd} F(Natural) \arrow[r, "F(hom)"] \arrow[d, "alg"'] & F(String) \arrow[d, "alg'"] \\ Natural \arrow[r, "hom"'] & String \end{tikzcd}$$

Figure 7:

$$\begin{tikzcd} F(Natural) & F(String) \arrow[l, "F(hom')"'] \\ Natural \arrow[u, "coalg"] & String \arrow[l, "hom'"] \arrow[u, "coalg'"'] \end{tikzcd}$$

Figure 8:

E.g.

$$
\begin{tikzcd}
F(Natural) & F(String) \arrow[l, "F(hom')"'] \\
Natural \arrow[u, "coalg"] & String \arrow[l, "hom'"] \arrow[u, "coalg'"']
\end{tikzcd}
$$

Figure 9:

```haskell
coalg :: Natural -> F Natural
coalg 0 = Zero
coalg n = Succ (n - 1)
```

E.g.

$$
\begin{tikzcd}
F(Natural) & F(String) \arrow[l, "F(hom')"'] \\
Natural \arrow[u, "coalg"] & String \arrow[l, "hom'"] \arrow[u, "coalg'"']
\end{tikzcd}
$$

Figure 9:

```haskell
coalg :: Natural -> F Natural
coalg 0 = Zero
coalg n = Succ (n - 1)
```

```haskell
coalg' :: String -> F String
coalg' "!" =
  Zero
coalg' ('Q':'U':'A':'C':'K':xs) =
  Succ xs
```

E.g.

$$
\begin{tikzcd}
F(Natural) & F(String) \arrow[l, "F(hom')"'] \\
Natural \arrow[u, "coalg"] & String \arrow[l, "hom'"] \arrow[u, "coalg'"']
\end{tikzcd}
$$

Figure 10:

```haskell
hom :: Natural -> String
hom n = timesN n "QUACK" ++ "!"
```

E.g.

$$
\begin{tikzcd}
F(Natural) & F(String) \arrow[l, "F(hom')"'] \\
Natural \arrow[u, "coalg"] & String \arrow[l, "hom'"] \arrow[u, "coalg'"']
\end{tikzcd}
$$

Figure 10:

```haskell
hom :: Natural -> String
hom n = timesN n "QUACK" ++ "!"
```

```haskell
hom' :: String -> Natural
hom' str =
  (fromIntegral (length str) - 1)
    `div` 5
```

E.g.

$$\begin{tikzcd} F(Natural) & F(String) \arrow[l, "F(hom')"'] \\ Natural \arrow[u, "coalg"] & String \arrow[l, "hom'"] \arrow[u, "coalg'"'] \end{tikzcd}$$

Figure 10:

```haskell
hom :: Natural -> String
hom n = timesN n "QUACK" ++ "!"
```

```haskell
hom' :: String -> Natural
hom' str =
  (fromIntegral (length str) - 1)
    `div` 5
```

```haskell
> (hom' "QUACKQUACK!", coalg' "QUACKQUACK!")
(2, Succ "QUACK!")
> (coalg $ hom' "QUACKQUACK!", fmap hom' $ coalg' "QUACKQUACK!")
(Succ 1, Succ 1)
```

The unique homomorphism

$$\forall \mathit{algebras}\ \exists \mathit{hom}\ : \mathit{carrier\ of\ algebra} \rightarrow \mathit{TermF}$$

$$\begin{tikzcd}
F\ TermF & F(Natural) \arrow[l, "F(hom')"'] \\
TermF \arrow[u, "unRoll"] & Natural \arrow[l, "hom'"] \arrow[u, "coalg"']
\end{tikzcd}$$

Figure 11:

The unique homomorphism

$$\forall algebras\ \exists hom\ :\ carrier\ of\ algebra \rightarrow TermF$$

$$\begin{tikzcd}
F\ TermF & F(Natural) \arrow[l, "F(hom')"'] \\
TermF \arrow[u, "unRoll"] & Natural \arrow[l, "hom'"] \arrow[u, "coalg"']
\end{tikzcd}$$

Figure 11:

```
type TermF = InitF
unRoll :: TermF -> F TermF
```

The unique homomorphism

$$\forall \textit{algebras}\ \exists \textit{hom}\ : \textit{carrier of algebra} \rightarrow \textit{TermF}$$

$$\begin{tikzcd}
F\ TermF & F(Natural) \arrow[l, "F(hom')"'] \\
TermF \arrow[u, "unRoll"] & Natural \arrow[l, "hom'"] \arrow[u, "coalg"']
\end{tikzcd}$$

Figure 11:

```haskell
type TermF = InitF
unRoll :: TermF -> F TermF
```

```haskell
hom' :: Natural -> TermF
```

The unique homomorphism

$$\forall algebras\ \exists hom\ : carrier\ of\ algebra \rightarrow TermF$$

```
Roll :: F TermF -> TermF
```

$$\begin{tikzcd}
F\ TermF \arrow[d, "Roll"'] & F(Natural) \arrow[l, "F(hom')"'] \\
TermF & Natural \arrow[l, "hom'"] \arrow[u, "coalg"']
\end{tikzcd}$$

Figure 12:

The unique homomorphism

$$\forall algebras\ \exists hom\ \ : carrier\ \ of\ \ algebra \rightarrow TermF$$

$$\begin{tikzcd}
F\ TermF \arrow[d, "Roll"'] & F(Natural) \arrow[l, "F(hom')"'] \\
TermF & Natural \arrow[l, "hom'"] \arrow[u, "coalg"']
\end{tikzcd}$$

Figure 12:

```
Roll :: F TermF -> TermF
```

```
hom' :: Natural -> TermF
```

The unique homomorphism

$$\forall \mathit{algebras}\ \exists \mathit{hom}\ : \mathit{carrier\ of\ algebra} \rightarrow \mathit{TermF}$$

$$\begin{tikzcd}
F\ TermF \arrow[d, "Roll"'] & F(Natural) \arrow[l, "F(hom')"'] \\
TermF & Natural \arrow[l, "hom'"] \arrow[u, "coalg"']
\end{tikzcd}$$

Figure 12:

```haskell
Roll :: F TermF -> TermF
```

```haskell
hom' :: Natural -> TermF
```

```haskell
hom' = Roll . fmap hom' . coalg
```

The unique homomorphism

$$\forall \mathit{algebras}\ \exists \mathit{hom}\ : \mathit{carrier\ of\ algebra} \rightarrow \mathit{TermF}$$

$$\begin{tikzcd}
F\ TermF \arrow[d, "Roll"'] & F(Natural) \arrow[l, "F(hom')"'] \\
TermF & Natural \arrow[l, "hom'"] \arrow[u, "coalg"']
\end{tikzcd}$$

Figure 12:

```
Roll :: F TermF -> TermF
```

```
hom' :: Natural -> TermF
```

```
hom' = Roll . fmap hom' . coalg
```

```
ana :: Functor f
    => (a -> f a) -> a -> Fix f
ana coalg =
  Roll . fmap (ana coalg) . f
```

Whilst we're here

```
cata :: Functor f => (f a -> a) -> InitF -> a
cata alg = alg . fmap (cata alg) . unRoll

ana :: Functor f => (a -> f a) -> a -> TermF
ana coalg = Roll . fmap (ana coalg) . coalg
```

Whilst we're here

```haskell
cata :: Functor f => (f a -> a) -> InitF -> a
cata alg = alg . fmap (cata alg) . unRoll

ana :: Functor f => (a -> f a) -> a -> TermF
ana coalg = Roll . fmap (ana coalg) . coalg
```

```haskell
cata alg' $ ana coalg' $ hom 3
> "QUACKQUACKQUACK!"
```

Whilst we're here

```haskell
cata :: Functor f => (f a -> a) -> InitF -> a
cata alg = alg . fmap (cata alg) . unRoll

ana :: Functor f => (a -> f a) -> a -> TermF
ana coalg = Roll . fmap (ana coalg) . coalg
```

```haskell
cata alg' $ ana coalg' $ hom 3
> "QUACKQUACKQUACK!"
```

```haskell
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo alg coalg = alg . fmap (hylo alg coalg) . coalg
```

Whilst we're here

```
cata :: Functor f => (f a -> a) -> InitF -> a
cata alg = alg . fmap (cata alg) . unRoll

ana :: Functor f => (a -> f a) -> a -> TermF
ana coalg = Roll . fmap (ana coalg) . coalg
```

```
cata alg' $ ana coalg' $ hom 3
> "QUACKQUACKQUACK!"
```

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> a -> b
hylo alg coalg = alg . fmap (hylo alg coalg) . coalg
```

```
hylo alg' coalg' $ hom 3
> "QUACKQUACKQUACK!"
```